Master in High Performance Computing

FPGA in HPC: High Level Synthesys of OpenCL kernels for Molecular Dynamics

Supervisors:
Stefano Cozzini,
Giuseppe Piero Brandino

Candidate:
Paolo Gorlani

3rd edition
2016–2017

Contents

Introduction

The overall goal of this thesis is to evaluate the feasibility of FPGA based computer system in HPC. This works is performed within ExaNeSt, an EU funded project which aims to develop and prototype energy efficient solutions for the production of exascale-level supercomputers.

As the matter of fact, the current computer architectures need to be re-thought in order to reach the exascale performance. Scale current technologies will make the power consumption so large, that to run and maintain such systems will be technologically and economically too much demanding.

Let's take for example the system TSUBAME3.0, the #1 in the Green500 list of June 2017, which has an energy efficiency of 14.11 GFLOPS/Watt. If we assume to be able to scale this system without loss of efficiency to reach 1 exaFLOPS, the resulting system will have a power consumption of 70.9 MW, 500 times more than the actual power (142 kW).

This makes clear that the power consumption issues demands the study of a new class of HPC systems based on energy efficient solutions.

The ExaNeSt project is trying to satisfy the energy efficiency constrain using a Multiprocessor System-on-Chip, which embeds on the same package ARM cores and a FPGA. These technologies are designed with special attention to power consumption. The project research area are interconnection network, storage and cooling systems. ExaNeSt is involved also in the early development phase of scientific software applications because this provides hints on how to design the hardware architecture. This is called co-design approach.

This thesis investigates the High Level Synthesis of OpenCL kernels. We studied the design and the performance of OpenCL kernel functionality brought into the programmable logic of a FPGA-based device. We apply this study to offload the compute intensive part of Molecular Dynamic codes into the FPGA.

In the rest of the chapter, we give a short overview of key concepts used in our work. Namely, we briefly review the basic features of a FPGA and we clarify the meaning of High Level Synthesis. The main aspects of OpenCL are presented also. Finally, we close the chapter with a short introduction of the computational method used, the Molecular Dynamics.

FPGA

Field Programmable Gate Arrays are semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLB in Figure 1) connected via programmable links. These devices make possible to develop compute functionality directly at the silicon level managing the connections of the Configurable Logic Blocks.

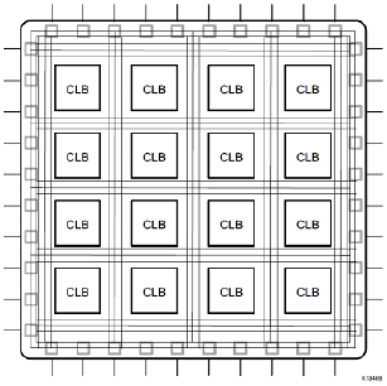


Figure 1: FPGA[8].

Hardware Description Language are used to describe the structure and behavior of digital circuits that can be implemented into the FPGA. The major HDL are VHDL and Verilog, they represent a circuit with Register-Transfer Level abstraction, which models a digital circuit as a flow of digital data between registers and logical operations performed on them.

Hardware Description Languages cannot be considered as usual programming languages (as C, C++, Python, Fortran ...). HDLs do not provide a list of instruction to be executed, they describe a digital hardware which is able to process the desired signals.

So, it is not possible to simply translate the programming languages instructions into HDL statements. The usage of High Level Synthesis is needed in order to get a hardware design from a program code.

High Level Synthesis

The High Level Synthesis is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior[5].

A High Level Synthesis software takes a function written in a programming language and produces a hardware description that can be implemented in FPGAs. In our thesis, we employed Vivado HLS. This tool makes possible to synthesize C, C++, SystemC and OpenCL functions in Verilog and VHDL. In particular, we studied the synthesis of kernels written in the OpenCL programming language.

OpenCL

Open Computing Language (OpenCL) is a framework for heterogeneous parallel computing platforms maintained by the industrial consortium Khronus Group. OpenCL defines a *programming language* (derived from C) and a series of *Application Programming Interfaces* for to manage the code execution. The OpenCL programming language is focus on the execution of many threads in parallel, it is applicable to different kind of computing *devices* as graphic cards or other kinds of accelerators. These devices are integrated in a *host* computer system that manages them thought the OpenCL API. Usually, the host computer system and the computing device do not share the same main memory space. OpenCL API provide also functions to manage transactions between the host and the device.

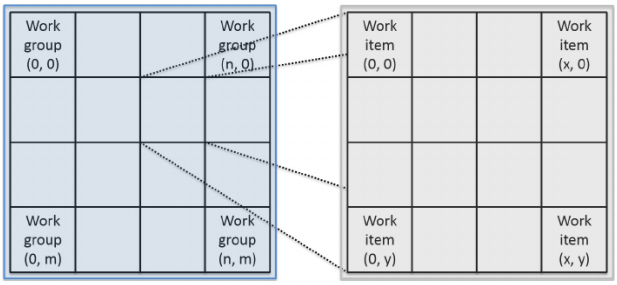


Figure 2: OpenCL kernel work item index scheme[10].

The functions written in the OpenCL language that run on the computing device are said *kernels*. The instructions in a kenel are executed independently by many *work-items*. A work-item is identified by a multidimensional index. A fixed number of work-items are grouped into a *work-group*. Also, work-groups are identified by a multidimensional index. This system of indexes is useful to distribute the work between the work-items in a block fashion way.

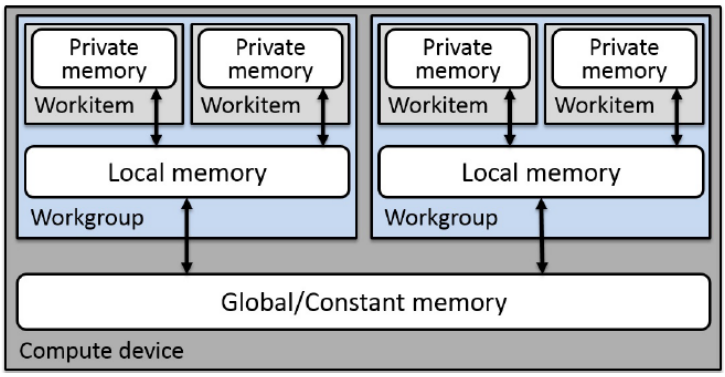


Figure 3: OpenCL memory model[10].

Moreover, OpenCL programming language defines the following memory model for data declared in kernels:

- the variables in the *global* memory context can be accessed by every work-item of every work-group,
- the *local* memory context contains variables shared between the work-items of the same work-group;
- the *private* memory context includes variable which are private with-in each work-item.

Molecular Dynamic

Molecular dynamics (MD) is a numerical simulation for studying the movements of particles (atoms or molecules) that can interact between each other for a fixed period of time. Particles interact through forces that can be calculated using interatomic potentials or molecular mechanics force fields. Given the total force acting on each particle, the numerical solution of Newton's equations of motion provides the dynamic evolution of the system. Molecular Dynamics is applied to chemical physics, materials science and bio-molecular modeling.

In Chapter 3, we analyze the synthesis of an OpenCL kernel that computes interatomic forces using the Lennard-Jones potential. We decide to concentrate our study on this operation because it is the most time-consuming part of Molecular Dynamics simulations.

Chapter 1

Hardware and software platform

In this chapter, we introduce the hardware and software platform adopted for this work. First of all, we describe the Xilinx Zynq Ultrascale+ device architecture and the AXI protocol, which manages its internal communication. Then, we show the software workflow that allows to port in the device an OpenCL kernel. In the last section, we consider the programming paradigm adopted to write OpenCL code that can be efficiently synthesized for the FPGA.

1.1 Hardware kit

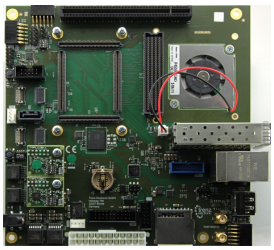
The goal of the ExaNeSt project is to develop a board, which contains four Xilinx Zynq Ultrascale+ chips. During the design finalization and production of the board, the ExaNeSt project employs the *Trenz Starter Kit 808*[1] as testbed. Such a board is composed of

- a Trenz TE0808-03 module equipped with a *Xilinx Zynq Ultrascale+ XCZU9EG* MPSoC and 2 GByte of DDR memory shown in Figure1.1[a];
- a Trenz TEBF0808-04A carrier board, which mounts the Trenz TE0808-03 module shown in Figure1.1[b].

The carrier board is placed inside a conventional ATX case (Figure1.1[c]). The power supply came from a standard ATX PSU. We employed this kit in order to test and analyze the capabilities of the Zynq Ultrascale+ MPSoC.

[1] https://wiki.trenz-electronic.de/display/PD/Starter+Kit+808

(a) TE0808-03 module



(b) Carrier board

(c) Board case

Figure 1.1: Trenz Starter Kit 808

1.2 Zynq Multiprocessor System-on-Chip

The XCZU9EG chip belongs to the Xilinx Zynq Ultrascale+ EG MPSoC production line. MPSoC stands for Multiprocessor System-on-Chip, which identifies a system-on-a-chip which uses multiple heterogeneous processors. In particular, Zynq devices are divided in two parts (Figure 1.2): the Processing System (PS) and the Programmable Logic (PL), which is the FPGA itself.

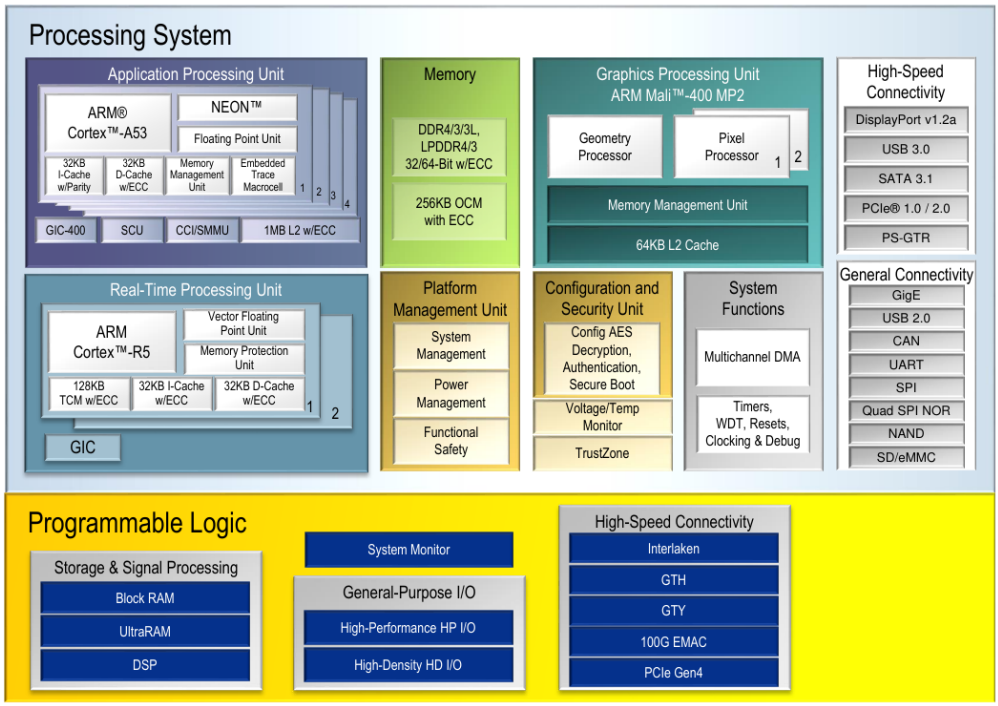


Figure 1.2: Xilinx Zynq Ultrascale+ EG

1.2.1 Programmable System (PS)

The Programmable System of our Zynq device contains, among the others, the following units:

- *Application Processing Unit* (APU). Quad-core ARM Cortex-A53 with 32KB L1 Cache and 1MB L2 Cache. ARM cores embed the NEON (advanced SIMD) technology for Single/Double Precision Floating Point computing. The APU is designed to carry out general purpose tasks as running an OS or performing floating-point computing. The clock frequency is up to 1.5 GHz.
- *Real Time Processing Unit* (RPU). Dual-core ARM Cortex-R5 with 32KB L1 Cache. The RPU is used for safety critical operations, which need to respond in a deterministic time. The clock frequency is up to 600 MHz.
- *Graphics Processing Unit* (GPU). ARM Mali-400 MP2 with 64KB L2 Cache. The GPU can be used exclusively for graphics and is not suitable for general computing tasks [2, p.44]. The clock frequency is up to 667 MHz.
- *Dynamic Memory Controller* (DDRC). A multi-ported memory controller that enables the PS and the PL to share the access to the DDR memory.

In this work, we employ the Application Processing Unit (APU) in order to drive the design in the Programmable Logic. The same task could be done also by the Real Time Processing Unit (RPU). We decide to use the APU because it is faster, its clock is higher and thus more appropriate for HPC. The RPU is best suited to execute tasks that require real-time processing.

1.2.2 Programmable Logic (PL)

The Programmable Logic contains the FPGA, an integrated circuit that can be programmed after fabrication. The FPGA contains various types of Configurable Logic Blocks, that can be wired together and configured in different ways. The main types of Configurable Logic Block are:

- Lookup Tables (LUT) are truth tables in which each combination of inputs returns an output. LUTs are capable to implement any Boolean logic function.
- Flip Flops (FF) store Boolean values. The input data is latched and passed to the output port on every clock cycle.

- Digital Signal Processing (DSP) are complex computational unit. The DSP are arithmetic logic units composed by three different stages: the pre-adder, the multiplier and the add/accumulate engine. These blocks implement function of the form $P = B(A + D) + C$ or $P+ = B(A + D)$.

- Block RAM (BRAM) are dual-port RAM modules, they can provide access to two different locations during the same clock cycle. Each BRAM can hold 18k or 36k bits. Not only BRAM can store data, LUTs can be used as 64-bit memories. LUT memories are fast because they can be instantiated in any part of the Programmable Logic, this reduce routing problem and increase performances.

Table 1.1 shows the total number of each Configurable Logic Block type in our device.

LUT	**FF**	**DSP**	**BRAM 18K**
548160	274080	2520	1824

Table 1.1: Available resources in the Programmable Logic of XCZU9EG.

1.2.3 Connections between PL and PS

The connections between PL and PS is one of the key point of our device. On Zynq Ultrascale+ devices, a total of 12 interfaces connect the PL with the PS. They are divided in Master and Slave interfaces.

The Master interfaces allows the PS to control the PL. They are `M_AXI_HPM0_LPD` and `M_AXI_HPM[0-1]_FPD` in Figure 1.3.

The Slave interfaces allows the PL to access the PS. These interfaces are usually provide connections to the DDR memory subsystem. This is the case of `S_AXI_HPC[0-1]_FPD`, `S_AXI_HP[0-3]_FPD` and `S_AXI_LPD` shown in Figure 1.3.

In our work, we use only the `S_AXI_HP[0-3]_FPD` interfaces because their specific function is to exchange (with high-performance) large data sets between the DDR memory and the PL. The interfaces `S_AXI_HPC[0-1]_FPD` have the same purpose, but they pass through the cache coherency interconnect (CCI) which manages cache coherency within the APU. We do not use them because we do not need this feature.

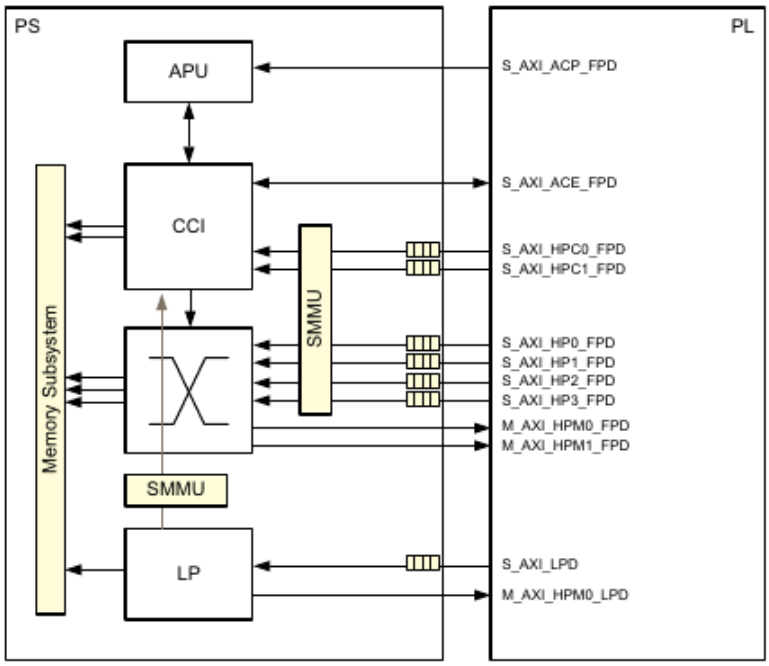


Figure 1.3: PL and PS interfaces

1.3 AXI protocol

In this section, we introduce the AXI protocol: an on-chip interconnect specification for the connection and management of functional units in a System-on-Chip. This protocol plays an important role in our work because it manages the communications between the PL and the PS. It allows to control the PL and to connect the PL to the DDR memory.

There are three types of AXI buses/interfaces:

- *AXI3/AXI4* for high-performance memory-mapped transactions. It allows burst of up to 16 for AXI3 and 256 for AXI4 data transfer cycles with just a single address phase. We use it to connect the functional blocks to the DDR memory.
- *AXI4-Lite* for simple, low-throughput, single memory-mapped transaction. We use it to control and to set the status of the functional blocks in PL.
- *AXI4-Stream* for high-performance stream (not mapped) transactions.

The AXI Master interfaces start AXI transactions to communicate with AXI

Slave interfaces. The transaction involve data exchange between the master and slave on multiple channels.

Memory-mapped transactions (performed by AXI4 and AXI-Lite) involves data transfers over a memory space mapped by addresses. Stream transactions do not involve a mapped memory space.

1.3.1 Burst memory access

The key feature of the AXI protocol are the burst memory transactions. *The master begins each burst by driving control information and the address of the first byte in the transaction to the slave. As the burst progresses, the slave must calculate the addresses of subsequent transfers in the burst[3]*, said beats. Having a single burst instead of multiple single memory transactions simplifies a lot the communication between the master and the slave. They need to exchange less control/synchronization messages improving the overall performance of transactions.

A burst is defined by the following characteristics.

- *Burst size* is the number of bytes transferred in each beat.
- *Burst length* is the total number of beats that form the burst.
- *Burst type* can be INCR, FIXED and WRAP. In a FIXED burst, the address is the same for every beat. This kind of burst is useful to empty FIFOs. In an INC burst, the address for each beat is an increment of the address of the previous one. The INC burst is used to access sequential memory locations. *A WRAP burst is similar to an INC burst, except that the address wraps around to a lower address if an upper address limit is reached. This burst type is used for cache line accesses[3].*

In our work, when we refer to burst, we mean the INC type. We use it in order to transfer sequential DDR memory locations form/to the PL. In Section 2.1.2, we present experimental results that show the importance of bursts.

Note. The burst size must not exceed the data bus width of either interface involved in the transaction. Otherwise, the transaction does not take place.

Note on Zynq. The internal Programmable System interfaces are AXI3 compliant, while the Programmable Logic interfaces are AXI4 compliant. The AXI3/AXI4 conversion takes place transparently in the Programmable System. For that reason, the maximum burst length for our device is 16 (i.e. the maximum burst length for AXI3 interfaces).

1.4 Software workflow

The implementation of kernels written in OpenCL programming language on a Zynq Ultrascale+ device involves three steps. Each step is performed by means of a software tool belonging to the *Xilinx Vivado Suite*[2]. We provide details of each step in the following sections.

1.4.1 Vivado High Level Synthesis

Vivado HLS performs the High Level Synthesis of existing codes written in C, C++, SystemC and OpenCL programming language. Vivado HLS takes a function written in such languages and produces a Register-Transfer Level design that models such function in a digital circuit. The synthesis of OpenCL kernels produces functional blocks having

- an AXI4-Lite Slave interface for block control and kernel arguments transmission (`s_axi_control` in Figure 1.4);
- an AXI4 Master interface for the connection to the DDR memory subsystem (`m_axi_gmem` in Figure 1.4).

These are the conventional interfaces for the OpenCL kernel synthesis. The designer cannot specify them otherwise. The synthesis of C, C++ and SystemC codes gives the possibility to choose more interface type and tune different parameters. More details on [7].

Vivado HLS supports version 1.0 of the OpenCL programming language. Moreover, it provides various directive in order to optimize the code synthesis. The most fundamental directives deal with the aspects of loop optimization and memory partition. The goal of the designer is to annotate the code with directives in order to produce a design with better performance. The directive list is available in [9]. These aspects are discussed in details in Section 1.5.

[2] `http://www.xilinx.com/products/design-tools/vivado.html`

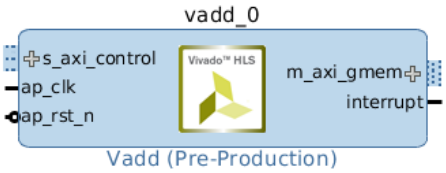


Figure 1.4: Synthesized functional block.

Vivado HLS at the end of the step produces a functional block (Figure 1.4) and a control driver written in C. The functional block needs then to be integrated within the Processing System by means of Vivado IDE. The driver is used to write a program running on the Programmable System that controls the functional block in the Programmable Logic.

Remark. The OpenCL programming language is no longer use beyond this step.

1.4.2 Vivado Integrated Design Environment

Vivado IDE is a design environment for Xilinx FPGAs. It creates, synthesizes, implement designs. We use Vivado IDE in order to integrate the functional block created in the previous step within the Zyqn Ultrascale+ device.

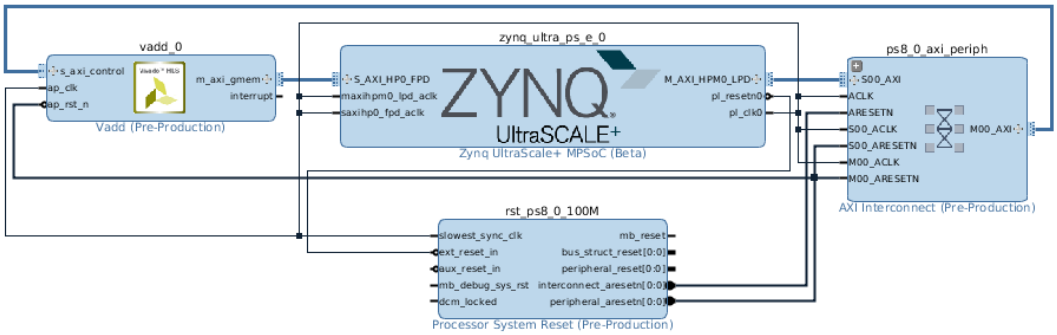


Figure 1.5: Project design.

After the creation of a valid design project (Figure 1.5), the following steps bring to the production of a bitstream, which contains information for the configuration of the FPGA logic.

- The *synthesis* is the process of transforming an RTL design into a FPGA logic representation. It starts from a Hardware Description Language listing

(Verilog or VHDL) and creates a lower level logic abstraction involving the Configurable Logic Blocks of the FPGA. The synthesis is performed for every block in the project design.

- The *implementation* places and routes the results of the block synthesis into the FPGA device resources while meeting the logical, physical, and timing constraints of the design. An important phase is the static timing analysis. This process calculates the timing delays of routes in order to verify if the clock constrains are meet.

- The *bitstream generation* produces the configuration bits that encode the result of the implementation. The bitstream is the data that is materially loaded into the FPGA logic.

The output of the Vivado IDE step is a Hardware Platform Specification File (`.hdf` extension). This file is a zip archive containing the bitstream, the functional block driver (created by Vivado HLS) and other various board configuration files. This archive is the starting point for board code development performed in the next step.

1.4.3 Xilinx Software Development Kit

Xilinx SDK is a software development environment based on Eclipse which makes possible to develop, debug and test C/C++ codes which control the functional blocks built in the Programmable Logic. These codes run on the Programmable System in the APU (our case) or in the RPU.

It must be noted that the proposed workflow do not use the OpenCL host code in order to control the logical blocks. The control code running on the Programmable System follows a completely different approach. Essentially, there is a reserved memory location for each functional block. The slave *AXI4-Lite* interface `s_axi_control` (in Figure 1.4) is used to read/write such memory location in order to get/set the control status of the block.

The block control is managed by four signals: start, ready, idle and done.

- The *start* signal is set to HIGH in order to start the computation on the functional block.

- The *idle* signal is asserted to LOW when the functional block is operating. It is HIGH when the computation is completed.

- The *ready* signal is asserted to HIGH when all the inputs are read and the functional block is ready to accept new inputs.
- The *done* signal is asserted to HIGH when the functional block has finished its operations and the outputs can be read.

These signals can be get/set through the C driver created by Vivado HLS. The start signal is set to HIGH invoking `XBlockname_Start` function. The status of ready, idle and done signals are returned by `XBlockname_IsDone`, `XBlockname_IsIdle` and `XBlockname_IsReady` functions. Where `Blockname` is the name of the functional block.

Furthermore, using Xilinx SDK it is possible to flash the FPGA if it is connected to the computer by a JTAG interface.

1.5 OpenCL from the FPGA prospective

The workflow described in the previous sections allows us to

- implement one or more *compute units* in the Programmable Logic;
- run a code in the Programmable System APU that is able to control *compute units* in the Programmable Logic.

We considered a *compute unit* as a digital circuit which performs the tasks of a working group of the corresponding OpenCL kernel. The following sections introduce how the OpenCL paradigm is applied to Zynq Ultrascale+ devices.

1.5.1 Computing aspects

The key concept in accelerator devices (as GPUs) is to have many independent working items running in parallel. It is different for FPGAs. Vivado HLS makes the execution of the working items sequential. Essentially, it puts the working items in a loop over their id. This loop can have variable or fixed bounds. In case of fixed boundaries Vivado HLS can optimize the size of local memories and provide the estimation of latency. The programmer can define the loop boundaries by setting the attribute `reqd_work_group_size`. This attribute specifies the size of the working group executed by a compute unit.

```
1 __attribute__ ((reqd_work_group_size(128, 64, 8)))
2 __kernel void foobar( ... )
3 {
4   int i = get_local_size(0);
5   int j = get_local_size(1);
6   int k = get_local_size(2);
7
8   ... // code to be executed
9
10 }
```

Listing 1.1: `reqd_work_group_size` example.

Listing 1.1 defines a kernel having the working group dimension of 128, 64 and 8 work items. Vivado HLS transparently trasforms this kernel in a function shown in Listing 1.2.

```
1 void foobar( ... )
2 {
3   for(int i = 0; i<128; ++i)
4     for(int j = 0; j<64; ++j)
5       for(int k = 0; k<8; ++k)
6       {
7         ... // code to be executed
8       }
9 }
```

Listing 1.2: Listing 1.1 transformation.

This code transformation makes loops very important. In fact, the main goal of the developer is to modify the kernel code in order to create efficient loops. The following characteristics are considered in loop optimization.

- *Count*: total number of loop iterations.
- *Iteration latency*: number of clock cycles spent for each iteration.
- *Total latency*: number of clock cycles spent for the entire loop.

Vivado HLS provides some directives in order to make loops run more efficiently. These directives deal with two standard loop optimization techniques: pipelining and unrolling. In the following sections, we explain the meaning of these techniques in High Level Synthesis.

Pipelining

Accordingly to [4, p.20]: *Pipelining is a digital design technique that allows the designer to avoid data dependencies and increase the level of parallelism in an algorithm hardware implementation. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. The only difference is the source of data for each stage. Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle .*

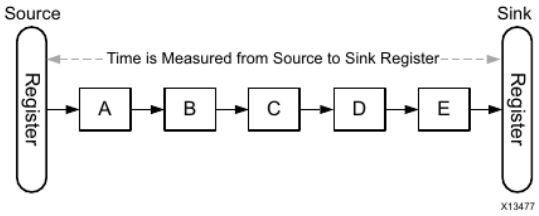


Figure 1.6: FPGA implementation without pipeline[4]. This design is made of 5 stages: A, B, C, D and E. There are just two registers, one at the beginning and one at the end. Just one signal per clock cycle can traverse all the stages.

This design makes possible to different loop iterations to be inside the pipeline at the same time. The number of clock cycles between the start time of consecutive iterations is said *Initialization Interval* (II). Vivado HLS tries to set II equals to 1 (i.e. a new iteration starts for each clock cycle). This is not always possible. The developer can also suggest the value of II.

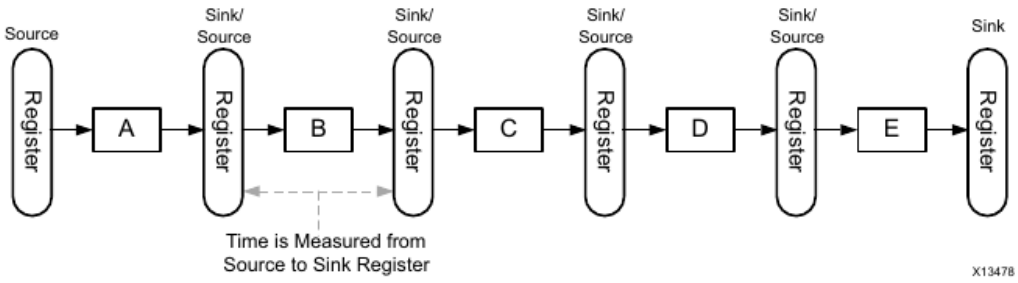


Figure 1.7: FPGA implementation with pipeline[4]. It is possible to pipeline the design in Figure 1.6 adding a register after each stage. In this way different signal can occupy different stage of the design in the same clock cycle.

Two kinds of attributes can be used to pipeline loops: `xcl_pipeline_loop` and `xcl_pipeline_workitems`.

The first attribute annotates `for` loops inside each work item. Listing 1.3 shows a code in which the `for` loop in line 5 is pipelined. It must be noted that the loop over all work-item ids (we remind that Vivado HLS substitute the parallelization over the work item with `for` loops as show in Listing 1.1 e 1.2.) is not executed in pipeline.

```
1 __attribute__ ((reqd_work_group_size(128, 1, 1)))
2 __kernel void foobar( ... )
3 {
4   __attribute__((xcl_pipeline_loop))
5     for(int i=0; i<8; ++i)
6     {
7       ... // code to be executed
8     }
9 }
```

Listing 1.3: `xcl_pipeline_loop` example.

If we want to pipeline the work items, it is possible to use `xcl_pipeline_workitems` attribute. This attribute annotates the instructions that are executed in pipeline in the work item loop created by Vivado HLS. For example, the instructions inside the curly bracket at line 6 in Listing 1.4 are executed in a pipeline.

```
1 __attribute__ ((reqd_work_group_size(128, 1, 1)))
2 __kernel void foobar( ... )
3 {
4   __attribute__((xcl_pipeline_workitems))
5   {
6     ... // code to be executed
7   }
8 }
```

Listing 1.4: `xcl_pipeline_workitems` example.

The effect of pipelinig is to reduce the total latency of loops. In fact, the total latency with and without pipeline can be computed as

$$total\ latency_{\ no\ pipe} \approx iteration\ latency \times loop\ count$$

$$total\ latency_{\ pipeline} \approx iteration\ latency + \left(II \times loop\ count \right)$$

The *iteration latency* in the two cases could be different, the insertion of multiple register can increase the *iteration latency* of the pipelined design. However, loop pipeling has been always advantageous in designs developed for our study.

Unrolling

The purpose of the loop unroll is to reduce the loop count and to execute different loop iteration at the same time. This can reduce latency and improve performance, but also consumes more FPGA resources. In fact, the stages needed by one iteration are replicated inside the FPGA.

The attribute `opencl_unroll_hint(n)` unrolls a loop with a factor of `n`. It is also possible unroll completely a loop using the `opencl_unroll_hint` attribute alone as shown in Listing 1.5.

```
1 __attribute__ ((reqd_work_group_size(128, 1, 1)))
2 __kernel void foobar( ... )
3 {
4
5   __attribute__((opencl_unroll_hint))
6     for(int i=0; i<8; ++i)
7     {
8       ...
9     }
10
11 }
```

Listing 1.5: `opencl_unroll_hint` example.

Very Important Note. It is not possible have nested pipelines and loops inside a pipeline. In the case there is a loop inside a pipeline it needs to be completely unrolled otherwise the pipeline cannot be created. The complete unroll is done in automatic by Vivado HLS, but this implies that the inner loops need to have a count known at compile time.

1.5.2 Memory aspects

It is important to note that the Programmable Logic and the Programmable System share the same DDR memory space. The developer does not need to transfer data between the host and the device memories using OpenCL API.

Global memory

The global memory is accessible by the Programmable System and all the compute units built in the Programmable Logic. This memory takes place in the *DDR chips*

on the Trenz TE0808-03 module (in Figure 1.1[a]) outside of the Zynq Ultrascale+ device.

We usually connect the master interface `m_axi_gmem` (Figure 1.4) of the compute units to the slave interface `S_AXI_HP[0-3]_FPD` (Figure 1.3) of the PS. In this way, the compute units can access the *DDR* memory passing through the PS memory subsystem.

This memory context is accessed using `global` pointers passed as kernel arguments. The width of the master interface `m_axi_gmem` is equal to the largest type define as `global` pointer in the kernel arguments. For example, the synthesis of the kernel in Listing 1.6 produces a compute unit in which the master interface is wide 32bit, the size of `int`. To maximize the data throughput, it is recommended to choose data types that have the same size of `S_AXI_HP[0-3]_FPD` width. In Section 2.1.1, we show experimental measure of the data throughput varying the argument datatype.

It is recommended to transfer data in bursts because this hides the memory access latency and improves bandwidth utilization. A dramatic improvement in data throughput is confirmed by the experimental results shown in Section 2.1.2. There are two ways to trig the bursts:

- using the `async_work_group_copy` function from the OpenCL programming language,
- requesting data from consecutive address locations in a pipelined loop.

The codes in Listing 1.6 and 1.7 do the same think. They read 128 elements of `in`, store the values in the local array `in_buff`, do some works, write the content of `out_buff` in `out`.

```
1  __attribute__ ((reqd_work_group_size(128, 1, 1)))
2  __kernel void foobar(__global int * in, __global int * out)
3  {
4    __local int in_buff[128], out_buff[128];
5
6    event_t e_in = async_work_group_copy(in_buff, in, 128, 0);
7    wait_group_events(1, &e_in);
8    ...
9    event_t e_out = async_work_group_copy(out, out_buff, 128, 0);
10   wait_group_events(1, &e_out);
11
12 }
```

Listing 1.6: `async_work_group_copy` burst example.

The synthesis of Listing 1.6 produces two pipelines, one for each `async_work_group_copy` function. Listing 1.7 produces only one pipeline. A pipeline can contain at most one burst write and one burst read, other memory transactions in the pipeline are not executed in burst.

```
1 __attribute__ ((reqd_work_group_size(128, 1, 1)))
2 __kernel void foobar(__global int * in, __global int * out)
3 {
4   __local int in_buff[128], out_buff[128];
5
6   __attribute__((xcl_pipeline_workitems))
7   {
8     in_buff = in[get_global_id(0)];
9     ...
10    out[get_global_id(0)] = out_buff;
11  }
12
13 }
```

Listing 1.7: Inferred burst example.

Local and Private memory

The `private` and `local` memory contexts identify variable that are stored in BRAM and LUT *inside the Programmable Logic*. These memories are very fast but their size is in the order of few MByte. This space is not small if compared with the size of the registers and the local memory in GPUs, which are in the order of some KByte.

The BRAMs can provide two memory accesses to different location at the same clock cycle. In the case in which an array needs to be accessed more than two times per clock cycle, it is possible to spread its content on multiple BRAMs. This practice is said array partition, it allows to improve data throughput inside the Programmable Logic making possible to access more data for each clock cycle. Various types of partitioning exist:

- *cyclic* is performed by putting consecutive array elements into different partitions in a round robin fashion;
- *block* is done by filling each partition with consecutive array elements;
- *complete* decomposes the array into individual elements.

It is possible to partition arrays using the `xcl_array_partition` attribute as shown in Listing 1.8. For example, `bufA` and `bufB` are spread on 4 cyclic/block partitions. Whereas, `bufC` is totally decomposed in 32 elements.

```
1 __attribute__ ((reqd_work_group_size(...)))
2 __kernel void foobar( ... )
3 {
4
5   int bufA[32] __attribute__((xcl_array_partition(cyclic,4,1)));
6   int bufB[32] __attribute__((xcl_array_partition(block,4,1)));
7   int bufC[32] __attribute__((xcl_array_partition(complete,1)));
8
9   ...
10
11 }
```

Listing 1.8: Partitioning example.

Vivado HLS analyses how arrays are accessed in order to partition them automatically. Sometimes, it fails. In that case the developer can partition them manually as in Listing 1.8.

Chapter 2

Performance analysis

This chapter describe the tests performed in order to understand how to write efficient codes for the Zynq Ultrascale+ device. The simplicity of the considered codes allows us to practice on the concepts introduced in the previous chapter and to build up an idea about the performance of our device.

2.1 Data throughput

We consider the data throughput as the rate at which data in the DDR memory can be transmitted and processed by the Programmable Logic of our device.

In the following section, we analyze the data throughput of different OpenCL kernels that performs the addition of two floating-point arrays and store the result in a third. We choose the array addition operation because it is usually a bandwidth bounded problem. In this way, we are able to study in detail which is the best approach to exploit the available bandwidth of our device.

We evaluate the data throughput measuring the *execution time* spent by Compute Units in the Programmable Logic to perform the array addition. The data throughput can be computed as

$$data\ throughput\ =\ \frac{data\ write\ +\ data\ read}{execution\ time}$$

In our tests, we performed the sum of 512 MB[1] sized floating point arrays. So, $data\ write$ = 512 MB and $data\ read$ = 1024 MB.

[1] 1 Megabyte = 1024 Kilobyte = 1024 * 1024 Bytes

2.1.1 Test argument datatype

In this section, we measure the data throughput varying the datatype of `global` pointers in the kernel arguments. The size of the datatype becomes the width of the Compute Unit master interface (`m_axi_gmem` in Figure 1.4). The master interface connects the Compute Unit to the DDR memory.

```
1 __kernel void __attribute__((reqd_work_group_size(WGSIZE, 1, 1)))
2 vadd(__global FTYPE *a, __global FTYPE *b, __global FTYPE *c)
3 {
4   __attribute__((xcl_pipeline_workitems))
5     c[get_global_id(0)] = a[get_global_id(0)] + b[get_global_id(0)];
6 }
```

Listing 2.1: Kernel code.

Listing 2.1 shows the considered OpenCL kernel. It is a plain array addition kernel which has been annotated with just two directive: the first at line 1 sets the work group size to `WGSIZE`, the second at line 4 creates a pipeline for the element summation. We kept the kernel code as simple as possible in order to observe the behavior of the Vivado HLS synthesis. It creates a pipeline with II (Initiation Interval) = 2, infers the burst write of `c` and reads `a` and `b` with single transactions (no burst).

We synthesized three different versions of this kernel varying the aggregate width of the vector type in kernel arguments. The size of the work group was changed in order to maintain constant the total number of work groups (see Table 2.5). In this way, the overhead to launch the work groups on the Compute Units is the same for all the three cases.

FTYPE	WGSIZE	Master interface width
`float`	2048	32
`float2`	1024	64
`float4`	512	128

Table 2.1: Kernel parameters.

We implemented the three Compute Unit in three different projects, all of them having the same design shown in Figure 2.2. In practice, we connected the master interfaces of 16 Compute Units to the same slave interface `S_AXI_HP0_FPD`. So, up to 16 CU share the DDR memory access through that slave interface. We set the width of `S_AXI_HP0_FPD` to its maximum (128 bit) and the Programmable Logic clock frequency to 100Mhz.

Results

For each project, we evaluate the data throughput increasing the number of used Compute Unit: 1, 2, 4 and 16. The results are shown in Table 2.2 and plotted in Figure 2.1. We can make the following observations.

- Data throughput increases linearly respect to the width of the CU master interface. It is maximum when the master interface have the same width of the slave interface, i.e. 128 bit.
- Data throughput do not increase further if more than 4 CU are used.

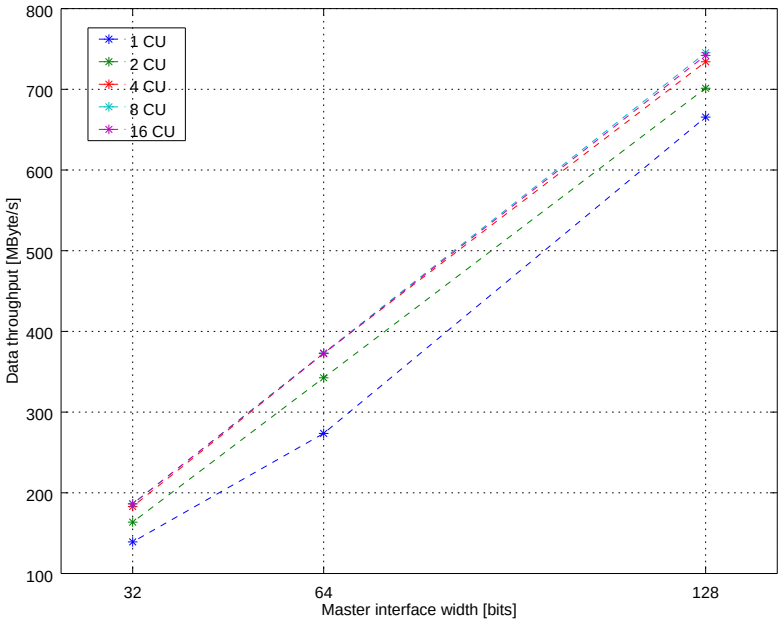


Figure 2.1: Plot of the data throughput varying master interface width and the number of used Compute Unit.

CU →	1	2	4	8	16
`float4` [128 bit]	665.56	701.32	734.13	745.35	742.06
`float2` [64 bit]	273.60	342.54	372.94	373.20	372.49
`float` [32 bit]	139.17	163.67	183.01	186.34	186.31

Table 2.2: Values expressed in **MB/s** of the data throughput varying master interface width and the number of used Compute Unit.

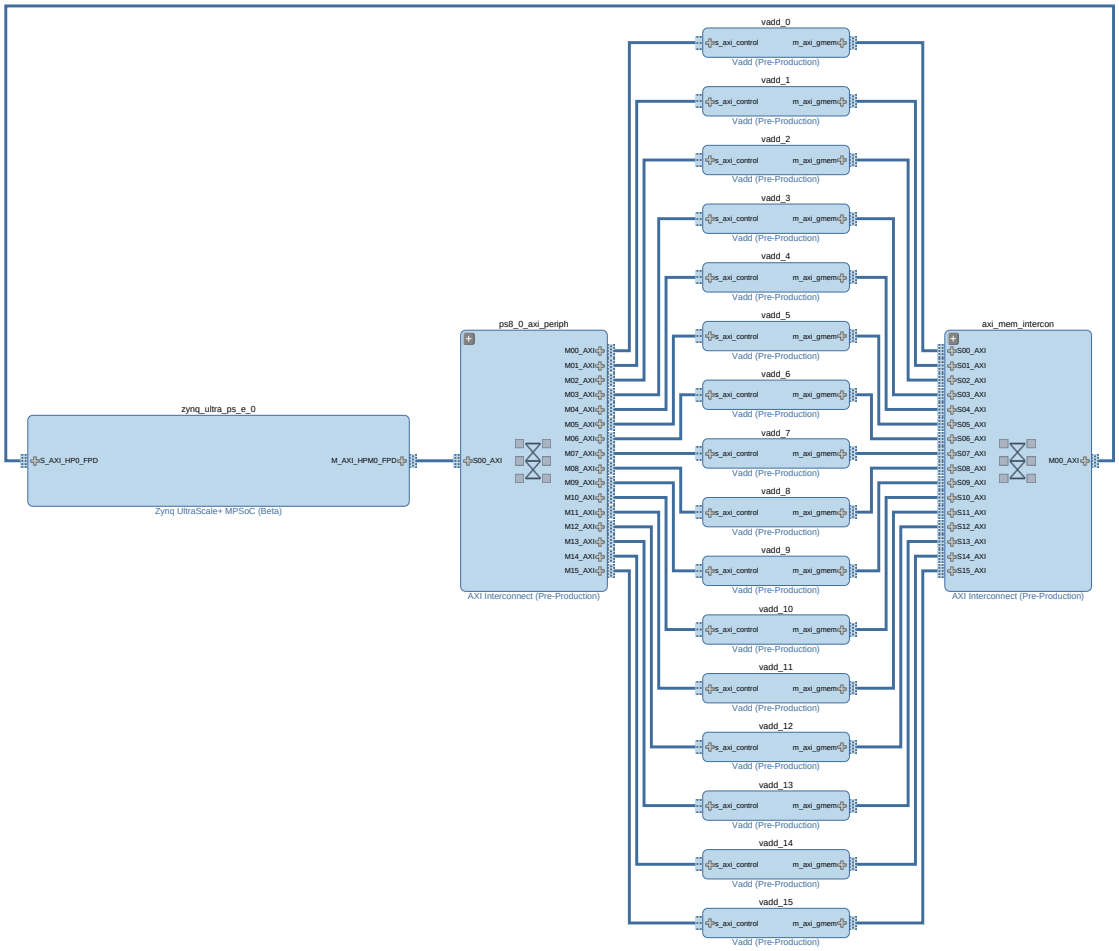


Figure 2.2: Project design: test argument datatype.

2.1.2 Test burst memory access

In the following section, we evaluate how data throughput changes when we enable or disable bursts. For these tests, we use all the four slave interface (S_AXI_HP[0-3]_FPD) in order to connect the Compute Units to the DDR memory. Furthermore, we test also the data throughput with different clock frequencies of the PL.

Listing 2.2 shows a kernel that performs all memory accesses in bursts. It operates on `float4` arrays because it is more efficient (as confirmed by tests in the previous section). This kernel contains two pipelines:

- the first at line 8 reads the values of `a` and stores them in `priv_a`,
- the second at lines 12-13 reads the values of `b` and stores them in `priv_b` and then stores the sum of `priv_a` and `priv_b` in `c`.

Vivado HLS infers a burst read for the first pipeline. For the second, a burst read and a burst write. In fact, Vivado HLS is able to infer bursts in at most one sequential read and one sequential write per pipeline.

```
1  __kernel void __attribute__ ((reqd_work_group_size(512,1,1)))
2  vadd(__global float4 *a, __global float4 *b, __global float4 *c)
3  {
4
5    float4 priv_a, priv_b;
6
7    __attribute__((xcl_pipeline_workitems))
8      priv_a = a[get_global_id(0)];
9
10   __attribute__((xcl_pipeline_workitems))
11   {
12     priv_b = b[get_global_id(0)];
13     c[get_global_id(0)] = priv_a + priv_b;
14   }
15
16 }
```

Listing 2.2: Burst access.

In order to evaluate the data throughput without bursts, we wrote another kernel modifying Listing 2.2 to avoid them. In sake of this, we simply reversed the sequential access of the arrays: we substituted `get_global_id(0)` with `LAST-get_global_id(0)` in lines 8,12 and 13. Where `LAST` is the index of the last element of the arrays. In this way, Vivado HLS is no more able to infer bursts.

In both kernels, Vivado HLS is able to create the pipelines with II (Initiation Interval) = 1. We synthesize these two kernels varying the clock frequency of the Programmable Logic to 100 , 200 and 300 MHz. For each frequency, we put the two kernels in two different projects. Their design is shown in Figure 2.4. There are 8 Compute Units in this design. We employed all the four `S_AXI_HP[0-3]_FPD` slave interfaces: two Compute Units are connected to one slave interface. So, at most 2 CU share the DDR memory access through the same slave interface.

Results

For each project, we evaluated the data throughput (with and without the burst) increasing the number of used Compute Units from 1 to 8. The results are shown in Table 2.3 and 2.4 and plotted in Figure 3.2. We can make the following observations.

- Burst memory transaction are fundamental to achieve good performance. The peak data throughput in this case is 8000 MB/s. Without burst, the peak performance drops to 2311 MB/s.
- The increase of the PL clock frequency produces a higher data throughput, especially in case of burst.
- In case of burst, higher is the PL clock frequency less CU are needed to get a high data throughput.

CU →	1	2	3	4	5	6	7	8
300 MHz	4402.1	6985.1	**8000.2**	7589.6	7706.2	7881.5	7701.4	7586.2
200 MHz	2988.5	5723.0	7403.0	7249.7	7129.8	**7510.9**	7353.7	7195.9
100 MHz	1534.6	3081.7	4619.4	6189.5	6275.8	6741.9	**7000.4**	6960.1

Table 2.3: Burst data throughput [**MB/s**] varying PL clock frequency and number of CU used.

CU →	1	2	3	4	5	6	7	8
300 MHz	1007.9	1650.6	1899.9	2216.5	2206.1	2231.5	2239.4	**2311.2**
200 MHz	1001.4	1616.7	1882.0	2199.9	2189.6	2215.6	2222.8	**2301.5**
100 MHz	529.56	1001.3	1515.1	1955.6	2032.7	2117.3	2200.3	**2275.5**

Table 2.4: Data throughput [**MB/s**] without burst varying PL clock frequency and number of CU used.

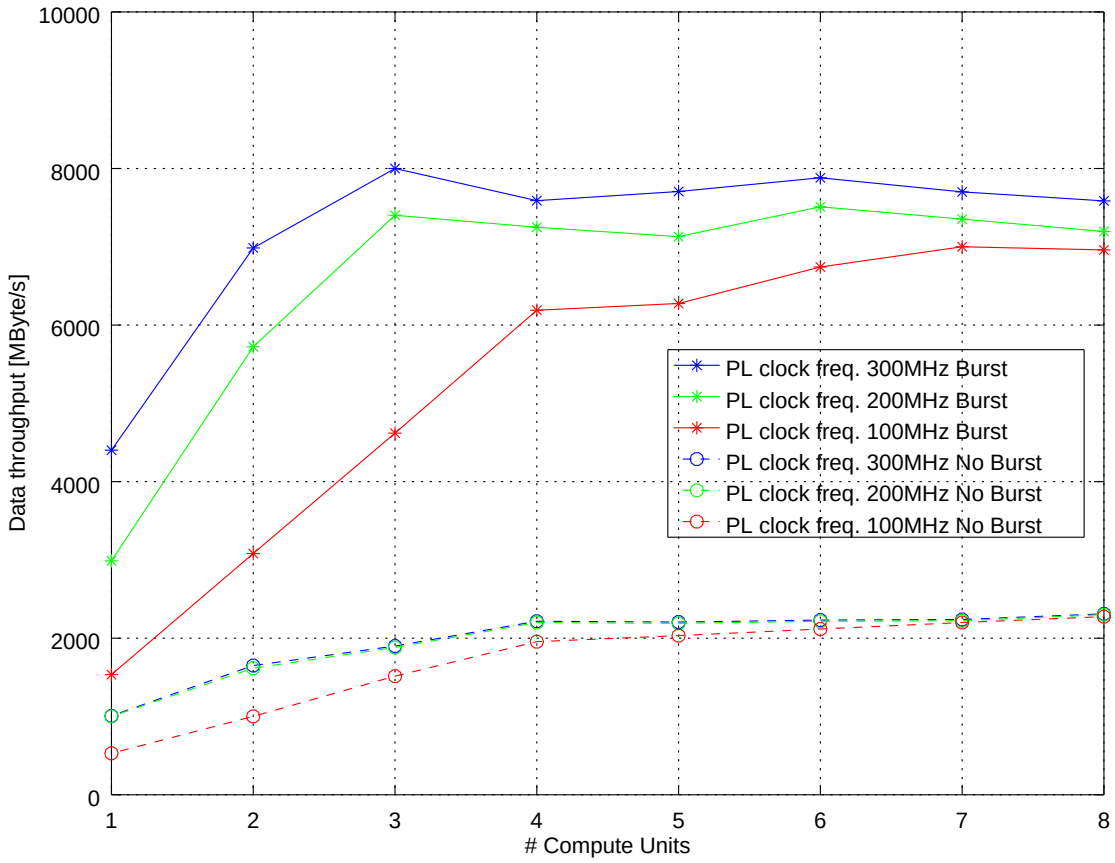


Figure 2.3: Data throughput with/without burst varying PL clock frequency.

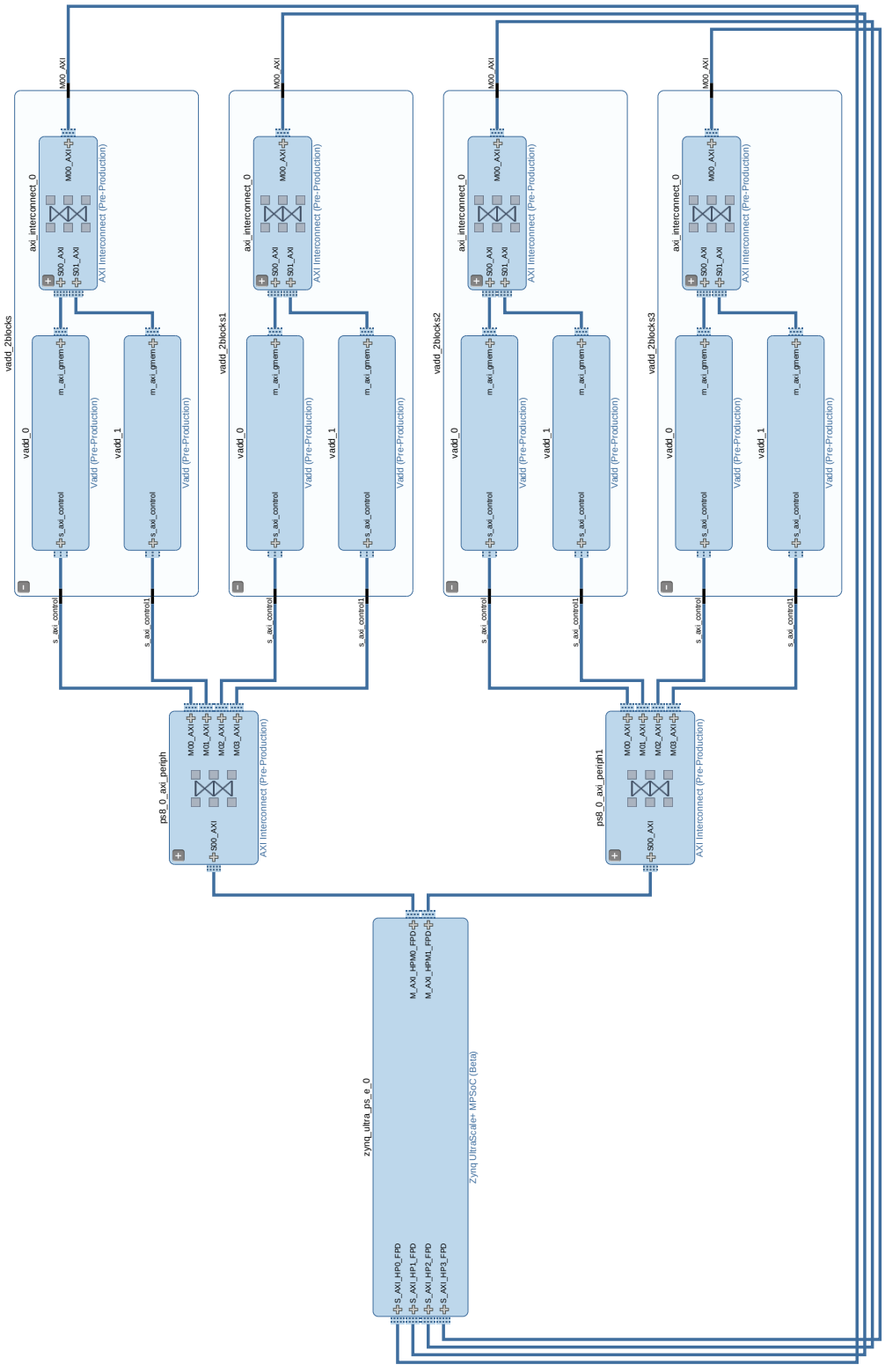


Figure 2.4: Project design: test burst memory access.

2.2 Floating Point Arithmetic

It is not easy to evaluate the floating-point peak performance of FPGAs. The Programmable Logic of the Zynq Utrascale+ devices lacks of specific units for floating-point computation. The floating-point operations need to be synthesized using the Configurable Logic Blocks. For example, floating-point additions are implemented using 2 DSP and a variable number of FF and LUT, depending on the PL clock frequency.

In order to have a *rough idea* about the floating-point performance, we followed the guideline in [6]. This white paper suggests to test a design that occupy most of FPGA resources performing floating-point additions at the maximum FPGA clock frequency. It is hard to successfully implement useful kernels in sake of to occupy most of the resources with a high clock frequency. Usually, the implementation fails for routing problems. Especially, when the synthesized kernel performs floating-point operations.

So, we developed a kernel (in Listing 2.3) that can be synthesized without problems using almost all the available DSPs, which are one of the key components of arithmetic operations. This kernel does a simple and useless operation: it sums recursively a float value with itself and an increasing index. In particular,

- lines 10-17 read the `N_FLOAT` float values in burst, and store them in the private array `p[]`;
- lines 19-23 perform the floating point computation: they sum recursively `N_FLOP` times the values of `p[]` with themselves and the index `r`;
- lines 26-33 burst write the content of `p[]` in the global memory.

It is possible to increase the number of floating point operations easily changing the `N_FLOP` parameter. Moreover, it is possible to maintain constant the number of floating-point addition units varying the `UNROLL` parameter.

N_FLOP	UNROLL	used DSP
39	32	**2496** (99%)
78	16	**2496** (99%)
624	2	**2496** (99%)

Table 2.5: Kernel parameters and PL resource usage.

Three different Compute Units are synthesized and implemented in three projects. We choose three couples of `N_FLOP` and `UNROLL` values (Table 2.5) in order to

keep constant the number of floating-point addition units (and as a consequence the number of DSPs).

The project design is very simple, there is only one Compute Unit connected to the slave interface S_AXI_HP0_FPD, as shown in Figure 1.5. The Programmable Logic clock frequency is 300 MHz.

```
1 __kernel void __attribute__ ((reqd_work_group_size(1, 1, 1)))
2 flop(__global float4 *g, int enable)
3 {
4
5   float p[N_FLOAT]
6     __attribute__((xcl_array_partition(cyclic,UNROLL,1)));
7
8   int l1_it = (enable==0)? 0 : N_FLOAT;
9
10  __attribute__((xcl_pipeline_loop))
11  load:for(int i=0; i < N_FLOAT/4; ++i)
12  {
13    p[4*i]   = g[i].x;
14    p[4*i+1] = g[i].y;
15    p[4*i+2] = g[i].z;
16    p[4*i+3] = g[i].w;
17  }
18
19  __attribute__((xcl_pipeline_loop))
20  l1:for(int I=0; I < l1_it; I+=UNROLL)
21    l2:for(int i=0; i < UNROLL; ++i)
22      l3:for(int r=0; r < N_FLOP; ++r)
23        p[I+i] = p[I+i] + ((float)r);
24
25
26  __attribute__((xcl_pipeline_loop))
27  write:for(int i=0; i < N_FLOAT/4; ++i)
28  {
29    g[i].x = p[4*i]  ;
30    g[i].y = p[4*i+1];
31    g[i].z = p[4*i+2];
32    g[i].w = p[4*i+3];
33  }
34
35 }
```

Listing 2.3: Kernel code

Results

We measured the execution times of the three kernel versions for a fixed size problem. We compute the FLoating-point Operation per Second as

$$FLOPS = \frac{number\ of\ float\ adds}{execution\ time}$$

the results are shown in the 3^{rd} column of Table 2.6.

Moreover, we measure execution times of the kernels performing just the memory transactions (without computation). This let us to measure the data transaction time in order to estimate the FLOPS excluding the data communication. This estimation considers only the floating-point computation performed by the PL, when data is already available in PL internal memory. These results are shown in the 4^{th} column of Table 2.6.

We have the following observations.

- More number of operations performed for each input (i.e. `N_FLOP`) produces better FLOPS performance. This shows that the limiting factor is the data throughput.
- The performance computed excluding the memory transaction are constant as the number of floating-point addition units.

We want to stress out the fact that the measured performance are related to the particular design developed. These results can give only a *rough idea* about the performance of our device. We believe, that it would be very hard to get performance of the same magnitude solving useful real-world problems.

`N_FLOP`	`UNROLL`	overall GFLOPS	compute only GFLOPS
39	32	21.5	347.3
78	16	40.5	347.1
624	2	178.3	347.3

Table 2.6: Performance results.

Chapter 3

Molecular Dynamics

In this chapter, we analyze the implementation of the Molecular Dynamics force compute kernel into the Programmable Logic of our Zynq Ultrascale+ device. Molecular Dynamics simulation algorithm involves many steps. We decide to study specifically the force computation step for the following reasons.

- It is performed at every time step of the simulation.
- It is compute intensive. A high order polynomial needs to be evaluated in order to get force values.
- It does not have a sequential pattern for most memory readings. In order to compute the forces acting on a particle, the positions of its neighbor particles need to be read. This implies a gather memory access pattern.

So, it is easy to understand that the force computation kernel needs to be the first to be ported and tested in novel architectures. We evaluate forces using the Lennard-Jones potential.

3.1 Lennard-Jones potential

The Lennard-Jones potential is a model that approximates the potential energy of a pair neutral atoms or molecules with given positions in space. It is expressed as

$$V(r) = 4\varepsilon \cdot \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{3.1}$$

where r is the distance between two particles, ε is the depth of the potential well and σ is the finite distance at which the inter-particle potential is zero. Given (3.1), the force between two particles can be computed as

$$\mathbf{F}(r) = -\nabla V(r) = -\frac{d}{dr}V(r)\hat{\mathbf{r}} = 4\varepsilon\left(12\,\frac{\sigma^{12}}{r^{13}} - 6\,\frac{\sigma^{6}}{r^{7}}\right)\hat{\mathbf{r}} \tag{3.2}$$

The total force acting on a particle is get by the summation of (3.2) computed for each particle neighbors. The neighbors are all the particles within a given distance of the considered particle. In fact, (3.2) tends rapidly to zero with increasing r.

As a starting point of our analysis, we considered the force computation implementation in MiniMD package.

3.2 MiniMD

MiniMD[12] is a simple and lightweight C++ code for molecular dynamics simulations. The algorithm and implementation mimics in a simpler way the same in LAMMPS[11], a well know and widely used MD package. In fact, miniMD is intended to be adapted to novel system architecture to test their performance.

MiniMD provides us a OpenCL kernel code that compute the forces and a series of C++ objects in order to generate a testbed for to provide initial values and validate the output.

The original kernel code is shown in Listings 3.1. It implements the following algorithm: for each particle in the system, compute the distance from each of its neighbors, if it is lower than a threshold compute the force and add its value to the total contribution.

3.3 Developed OpenCL kernels

In this section, we introduce a kernel developed in order to port the MiniMD OpenCL force kernel (Listing 3.1) in the Programmable Logic of our device.

During our study, we developed many different versions of this kernel. We discuss here in details only the one with the best performance. However, a lot of work has been done introducing much more complex approaches. But, they do not show any performance improvement respect to the described one.

Experimental measures in Chapter 2 confirm the importance to access global memory using bursts. So, we re-wrote the original MiniMD kernel (Listing 3.1) in order to read and write sequential data in burst. To achieve this purpose, we use the `async_work_group_copy` function and the `xcl_pipeline_workitems` attribute, as described in Section 1.5.2. This involve also the creation of local buffer in order to stores the data accessed in burst. These local buffers are partitioned in order to provide enough data per clock cycle. The resulting kernel is shown in Listing 3.2.

Then, we decided to modify further Listing 3.2 in order to unroll the loop over the compute intensive part, which evaluates the high order polynomial. In this way, we exposed more concurrency consuming more Programmable Logic resources. Moreover, this modification allows to add easily a flag to switch off the computation. In this way, we are able to evaluate the time taken by the memory transactions. The resulting kernel is shown in Listing 3.3.

In the following sections, we show the performance results of kernels in Listing 3.2 and 3.3, which are refereed respectively as *plain version* and *unroll version*. For both, Vivado HLS is able to synthesize all their pipelines with an II (Initiation Interval) = 1.

In order to evaluate the performance, we measure the time spent in the force computation for a system with 1048576 particles, the same for every test run. In order to validate the forces computed on the Programmable Logic, we evaluate the maximum absolute difference respect to the same computation performed on the APU using a MiniMD C++ function.

3.3.1 Plain version results

We synthesized the plain version kernel with three different PL clock frequencies: 100, 200 and 300 Mhz. We created one project for each frequency. The projects with a PL clock frequency of 100 and 200 MHz include 16 Compute Units. The project with a PL clock frequency of 300MHz includes only 4 Compute Units, because adding more CU makes the static time analysis to fail. In all the project, the CUs access to the global memory throw all the four slave interface `S_AXI_HP[0-3]_FPD`.

The time to compute the forces varying the number of used Compute Units are shown in Table 3.1 and plotted in Figure 3.1. The maximum absolute difference with the APU computation is `1.73226e-06`.

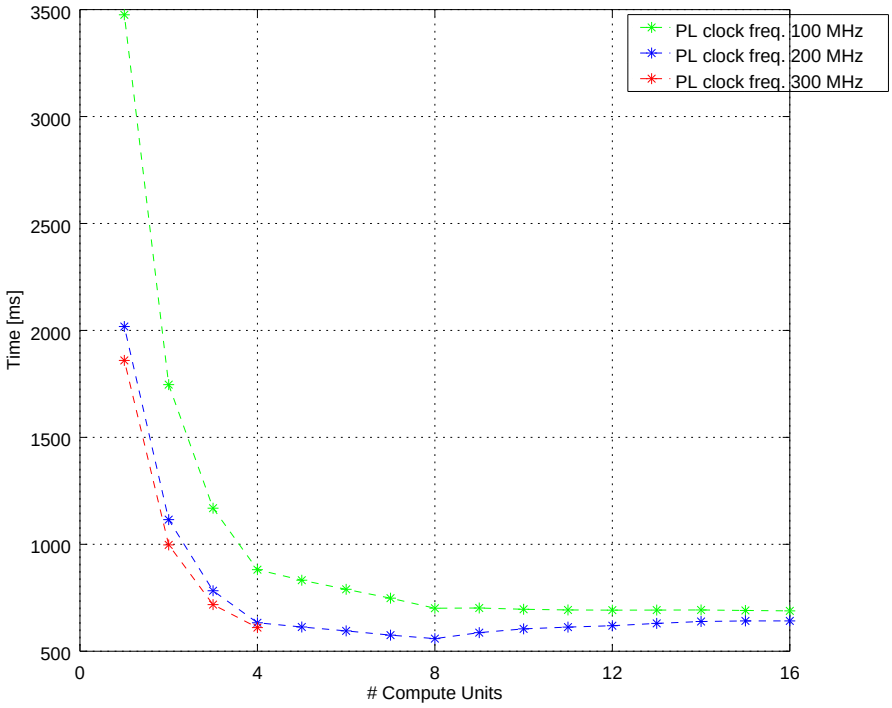


Figure 3.1: Force computation time for a 1048576 particle system varying the number of used CU and the PL clock frequency.

CU →	1	2	3	4	5	6	7	8
100MHz	3475.7	1746.6	1168.8	881.4	832.1	789.5	748.1	701.1
200MHz	2018.2	1115.4	782.9	633.1	613.4	595.7	575.3	**559.0**
300MHz	1859.7	997.2	718.0	**610.5**				

CU →	9	10	11	12	13	14	15	16
100MHz	702.1	696.2	693.1	692.1	692.5	693.0	690.0	**688.7**
200MHz	587.0	604.6	612.8	619.4	630.2	639.0	641.8	641.8

Table 3.1: Force computation time [**ms**] for a 1048576 particle system varying the number of used CU and the PL clock frequency.

3.3.2 Unroll version results

We synthesized the unroll version kernel with a PL clock frequency of 200 MHz. We were able to implement a project with at most 4 CUs with an unroll factor equal to 5. The unroll consumes more PL resource, so it was not possible to add more CUs. In order to confront the performance, we create another equal project with the plain version kernel. In these project, each CU is connected directly to one of the four slave interface `S_AXI_HP[0-3]_FPD`.

Table 3.2 show the solution time of the plain and unroll versions. We reported also the time of only the memory transactions. The maximum absolute difference with the APU computation is `2.48104e-06`.

The results clearly show that most of the time is spent on memory transaction. The unroll brings only a negligible improvement.

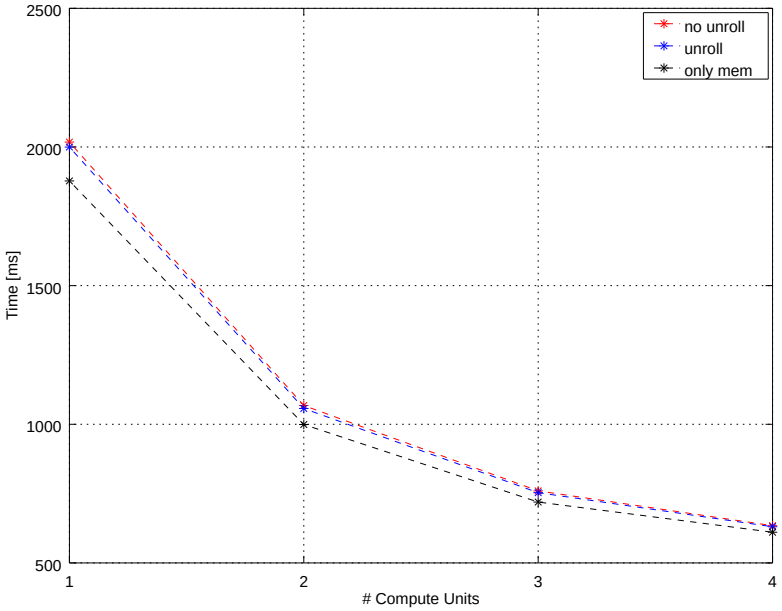


Figure 3.2: Plot of force computation times of Table 3.2

CU →	1	2	3	4
plain	2017.6	1067.7	759.8	634.3
unroll	1999.7	1056.9	753.4	630.4
mem. only	1877.6	999.2	719.5	610.7

Table 3.2: Force computation time [**ms**].

3.4 Listings

In the following section, we report the kernel considered in the previous part of the chapter. All the kernels take as input the following arguments.

- `x` a `float3` 1d-array containing the particle position.
- `numneigh` a `int` 1d-array containing the number of neighbors for each particle.
- `neighbors` a `int` 2d-array, it contains the indexes of each particle neighbors.
- `maxneighs` a `int` scalar expressing the maximum number of neighbors that is possible for a particle.
- `cutforcesq` a `float` scalar expressing the square of the distance above that it is possible to avoid computing the force because its value is negligible.

Its output is `f`, a `float3` array containing the force acting on each particle.

3.4.1 Original MiniMD kernel

```
1  typedef float MMD_float; typedef float3 MMD_floatK3;
2
3  __kernel void force_compute( __global MMD_floatK3* x,
4                               __global MMD_floatK3* f,
5                               __global int* numneigh,
6                               __global int* neighbors,
7                               int maxneighs,
8                               int nlocal,
9                               MMD_float cutforcesq )
10 {
11
12   int i = get_global_id(0);
13   if(i<nlocal)
14   {
15     __global int* neighs = neighbors + i;
16
17     MMD_floatK3 xi = x[i];
18     MMD_floatK3 fi = {0.0f,0.0f,0.0f};
19
20     for (int k = 0; k < numneigh[i]; k++)
21     {
22       int j = neighs[k*nlocal];
```

```
23        MMD_floatK3 delx = xi - x[j];
24
25        MMD_float rsq = delx.x*delx.x + delx.y*delx.y + delx.z*delx.z;
26
27        if (rsq < cutforcesq)
28        {
29          MMD_float sr2 = 1.0f/rsq;
30          MMD_float sr6 = sr2*sr2*sr2;
31          MMD_float force = 48.0f*sr6*(sr6-0.5f)*sr2;
32          fi += force * delx;
33        }
34
35      }
36
37      f[i] = fi;
38
39   }
40
41 }
```

Listing 3.1: MiniMD original force kernel code.

3.4.2 Plain version kernel

```
1  typedef float MMD_float; typedef float3 MMD_floatK3;
2
3  #define WG_SIZE (512)
4
5  __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))
6  __kernel void force_compute(  __global MMD_floatK3* x,
7                                __global MMD_floatK3* f,
8                                __global int* numneigh,
9                                __global int* neighbors,
10                               int maxneighs,
11                               int nlocal,
12                               MMD_float cutforcesq )
13 {
14
15   MMD_floatK3 xi, fi;
16   __local int local_numneigh[WG_SIZE]
17     __attribute__((xcl_array_partition(cyclic,2,1)));
18
19 // if(get_global_id(0) < nlocal)
20   {
21
22     __attribute__((xcl_pipeline_workitems))
23     {
24       xi = x[get_global_id(0)];
25       fi = 0.0f;
```

```
26      }
27
28      event_t e0 = async_work_group_copy(local_numneigh,
29                      numneigh + WG_SIZE*get_group_id(0), WG_SIZE, 0);
30
31      wait_group_events(1, &e0);
32
33      for (int k = 0; k < maxneighs; k++)
34      {
35
36        __local int neigh_idx[WG_SIZE]
37           __attribute__((xcl_array_partition(cyclic,4,1)));
38
39        event_t e1 = async_work_group_copy( neigh_idx,
40                        neighbors  + WG_SIZE*get_group_id(0) + k*nlocal,
41                        WG_SIZE, 0 );
42
43        wait_group_events(1, &e1);
44
45        __attribute__((xcl_pipeline_workitems));
46          if(k < local_numneigh[get_local_id(0)])
47          {
48
49            MMD_floatK3 delx = xi - x[neigh_idx[get_local_id(0)]];
50
51            MMD_float rsq = delx.x*delx.x
52                            + delx.y*delx.y + delx.z*delx.z;
53
54            if (rsq < cutforcesq)
55            {
56              MMD_float sr2 = 1.0f/rsq;
57              MMD_float sr6 = sr2*sr2*sr2;
58              MMD_float force = 48.0f*sr6*(sr6-0.5f)*sr2;
59              fi += force * delx;
60            }
61          }
62
63      }
64
65      __attribute__((xcl_pipeline_workitems))
66        f[get_global_id(0)] = fi;
67
68    }
69
70  }
```

Listing 3.2: Plain version kernel code.

3.4.3 Unroll version kernel

```
1  typedef float MMD_float; typedef float3 MMD_floatK3;
2
3  #define WG_SIZE (512)
4  #define UNROLL 5
5
6  __attribute__ ((reqd_work_group_size(WG_SIZE, 1, 1)))
7  __kernel void force_compute( __global MMD_floatK3* x,
8                               __global MMD_floatK3* f,
9                               __global int* numneigh,
10                              __global int* neighbors,
11                              int maxneighs,
12                              int nlocal,
13                              MMD_float cutforcesq,
14                              int enable )
15 {
16
17   MMD_floatK3 xi, fi;
18   MMD_floatK3 _fi[UNROLL];
19   __local int local_numneigh[WG_SIZE]
20      __attribute__((xcl_array_partition(complete,1)));
21
22   const int cpipe_count = (enable == 0) ? 0 : UNROLL;
23
24 // if(get_global_id(0) < nlocal)
25   {
26
27     __attribute__((xcl_pipeline_workitems))
28     {
29       xi = x[get_global_id(0)];
30       fi = 0.0f;
31       for(int K=0; K<UNROLL; ++K) _fi[K] = 0.0f;
32     }
33
34     event_t e0 = async_work_group_copy(local_numneigh,
35                    numneigh + WG_SIZE*get_group_id(0), WG_SIZE, 0);
36
37     wait_group_events(1, &e0);
38
39     for (int k = 0; k < maxneighs; k+=UNROLL)
40     {
41
42       __local int neigh_idx[UNROLL][WG_SIZE]
43          __attribute__((xcl_array_partition(complete,1)))
44          __attribute__((xcl_array_partition(cyclic,8,2)));
45
46       MMD_floatK3 delx[UNROLL]
47          __attribute__((xcl_array_partition(cyclic,5,1)));
```

```
48
49      event_t e1 = async_work_group_copy(neigh_idx[0],
50                    neighbors  + WG_SIZE*get_group_id(0) + k*nlocal,
51                    WG_SIZE, 0);
52
53      __attribute__((opencl_unroll_hint(UNROLL-1)))
54      for(int K=1; K<UNROLL; ++K)
55        if(K+k < maxneighs)
56          async_work_group_copy(neigh_idx[K],
57            neighbors  + WG_SIZE*get_group_id(0) + (K+k)*nlocal,
58            WG_SIZE, e1);
59
60      wait_group_events(UNROLL, &e1);
61
62      for(int K=0; K<UNROLL; ++K)
63        __attribute__((xcl_pipeline_workitems))
64          if(K+k < local_numneigh[get_local_id(0)])
65            delx[K] = xi - x[neigh_idx[K][get_local_id(0)]];
66
67      if(enable != 0)
68      __attribute__((xcl_pipeline_workitems))
69        for(int K=0; K<UNROLL; ++K)
70          if(K+k < local_numneigh[get_local_id(0)])
71          {
72            MMD_float rsq = delx[K].x*delx[K].x + delx[K].y*delx[K].y
                  + delx[K].z*delx[K].z;
73            if (rsq < cutforcesq)
74            {
75              MMD_float sr2 = 1.0f/rsq;
76              MMD_float sr6 = sr2*sr2*sr2;
77              MMD_float force = 48.0f*sr6*(sr6-0.5f)*sr2;
78              _fi[K] += force * delx[K];
79            }
80          }
81
82    }
83
84    __attribute__((xcl_pipeline_workitems))
85    {
86      for(int K=0; K<UNROLL; ++K)
87        fi += _fi[K];
88      f[get_global_id(0)] = fi;
89    }
90
91  }
92
93 }
```

Listing 3.3: Unroll version kernel code.

Conclusions

This thesis study spawned a seven months period, from May to December 2017. Our previous experience was in the field of scientific code development and did not include any topics related to FPGAs or digital circuit design. At the end of our work, we developed the following considerations.

- This topic have steep learning curve. In order to have a running code on the Programmable Logic of the Zynq Ultrascale+ device, a number of steps need to be performed. Each one leads to the configuration of various aspects. Understand these aspects involves the study of technical documentation conceived for digital circuit designer: the main users of FPGAs.
- An efficient port on FPGA of OpenCL kernels involves change completely their code. In our opinion, the OpenCL programming language is not well suited for FPGAs. In fact, the programming model of the High Level Synthesis of Vivado HLS is based on loops optimization (pipeline and unroll) and not on parallel working items. There is no convenience of using OpenCL compared to C. Indeed, Vivado HLS provides more optimization directives for the synthesis of C codes: specially for the definition of the master/slave interfaces.
- The main performance limiter of the Zynq Ultrascale+ device is the main memory bandwidth. Sequential memory transactions (in burst) are mandatory in order to exploit efficiently the low available bandwidth. Not all the computational applications can overcome efficiently this constraint.

The flexibility of the FPGA takes the form of a large number of aspects to consider. Understand and exploit successfully them is the challenge to win in order to utilize efficiently the FPGA great capabilities.

Bibliography

[1] Xilinx. *Zynq UltraScale+ Device Technical Reference Manual [UG1085].* https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

[2] Xilinx. *Zynq UltraScale+ MPSoC Embedded Design Methodology Guide [UG1228].* https://www.xilinx.com/support/documentation/sw_manuals/ug1228-ultrafast-embedded-design-methodology-guide.pdf

[3] ARM. *AMBA, AXI and ACE Protocol Specification.* http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf

[4] Xilinx. *Introduction to FPGA Design with Vivado HLS [UG998].* https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

[5] Wikipedia. *High-level synthesis.* https://en.wikipedia.org/wiki/High_Level_Synthesis

[6] Michael Parker. *Understanding Peak Floating-Point Performance Claims.* https://www.altera.com/en_US/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf

[7] Xilinx. *Vivado Design SuiteUser Guide: High-Level Synthesis [UG902].* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug902-vivado-high-level-synthesis.pdf

[8] Xilinx. *SDAccel Optimization Guide [UG1207].* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1207-sdaccel-optimization-guide.pdf

[9] Xilinx. *SDx Pragma Reference Guide [UG1253].*

[10] Tatsumi, Shunsuke, et al. *OpenCL-based design of an FPGA accelerator for phase-based correspondence matching.* Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.

[11] LAMMPS Molecular Dynamics Simulator `http://lammps.sandia.gov/`

[12] MiniMD from Mantevo package `https://mantevo.org/`